Princeton University

Mirror Machine Loss-Cones and Boundary Loss-Ratios in the Presence of Collisions and Steps in Finding Optimized Magnetic Field Geometries of the Magnetic Centrifugal Mass Filter

Dylan Mavrides

June - September, 2016

This project was done as a part of the 2016 Program in Plasma Science and Technology (PPST) at Princeton University - a full-time summer internship at the Princeton Plasma Physics Lab (PPPL) - with extensive guidance from graduate student Ian Ochs and with the support of executive committee member Professor Nathaniel J. Fisch.

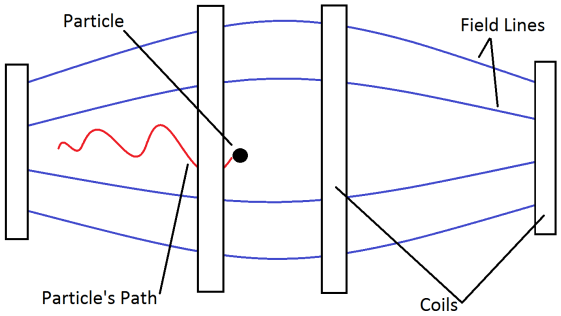
Thanks also goes to Professor Samuel Cohen, co-chair of the PPST executive committee, for his administrative help and devotion to the program.

Sponsored by the PEI Seibel Grand Challenges Program.

1 Project Background and Description

1.1 Basic Background Physics

The "magnetic mirror machine" is a plasma-confinement device which relies on a magnetic field consisting of two strong regions with a weaker region in between them. Taking advantage of conservation of the magnetic moment, charged particles with initial velocity conditions bounded by certain values will remain trapped in the interior of the device. In this way, high-energy particles can be used for plasma experiments and devices in a controlled manner.



Particles which begin in a region with lowest magnetic field strength B_i, velocity component parallel to that $v_{\parallel_i}$, and velocity component perpendicular to it $v_{\perp_i}$ can be described as having magnetic moment:

$$\mu = \frac{v_{\perp_i}^2}{B_i}$$

Using conservation of energy we can derive the condition for the particle to remain bounded by a magnetic field of strength $B_f > B_i$

$$KE_i < KE_f$$

$$KE_{\parallel_i} + KE_{\perp_i} < KE_{\parallel_f} + KE_{\perp_f}$$

Note that $KE_{\perp_f} = 0$

$$\frac{mv_{\parallel_i}^2}{2} + \frac{mv_{\perp_f}^2}{2} < \frac{mv_{\perp_f}^2}{2}$$

$$v_{\parallel_i}^2 + v_{\perp_i}^2 < v_{\perp_f}^2$$

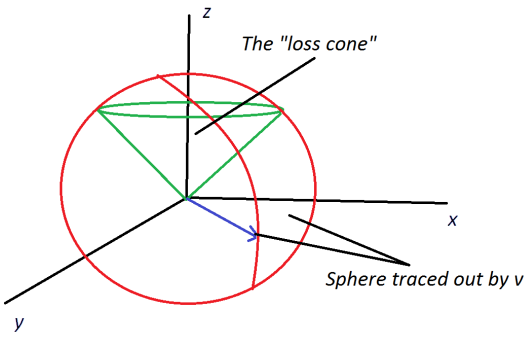
Since the magnetic moment is conserved, $\mu = \frac{v_{\perp_i}^2}{B_i} = \frac{v_{\perp_f}^2}{B_f}$ and thus $v_{\perp_f}^2 = \frac{B_f}{B_i} v_{\perp_i}^2$

$$v_{\parallel_i}^2 + v_{\perp_i}^2 < \frac{B_f}{B_i} v_{\perp_i}^2$$

Let the mirror ratio, $\frac{B_f}{B_i}$ be R_m

$$\frac{|v_{\|_i}|}{|v_{\perp_i}|} < \sqrt{R_m - 1} \tag{1.1}$$

From this equation, we can make a graph representing the possible velocities a particle can have in the mirror machine such that it is contained. Because of the 3-dimensional shape of this graph, it is known as the loss cone.

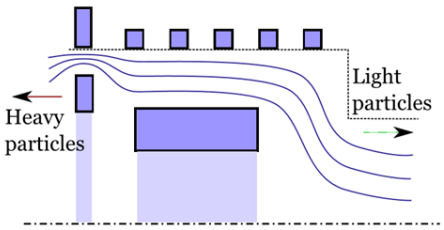


Let z be the perpendicular axis. Note that if a particle has a velocity value that falls within the region of the loss cone, it would not be contained by the mirror ratio describing the slope of the cone.

1.2 The Magnetic Centrifugal Mass Filter

The magnetic centrifugal mass filter (MCMF) is a device proposed by Abraham J. Fetterman and Nathaniel J. Fisch in 2011 which aims to use the magnetic mirror effect as well as a centrifugal effect resulting from an electrically driven rotation in order to to separate ions based on mass [1]. Since this device is less sensitive to differences in mass than other proposed devices, it could be used as a separation device for nuclear waste with significantly decreased risk of nuclear proliferation.

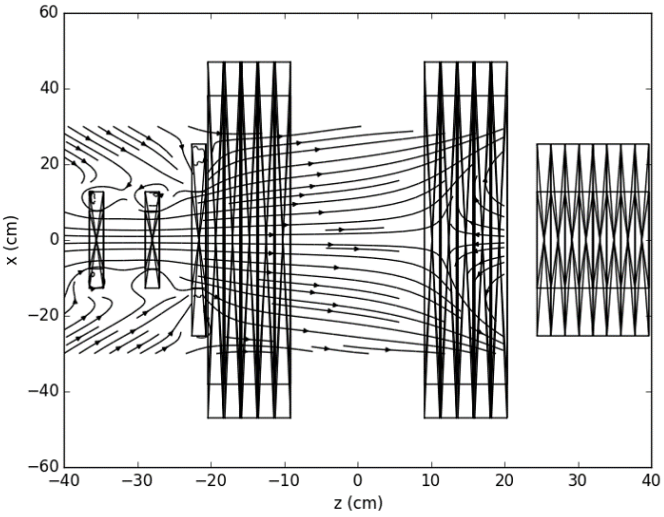
From Fetterman's thesis, a basic diagram of a cross section of the machine[2]:



A magnetic centrifugal mass filter. The solid lines indicate magnetic field lines, shaded squares indicate magnetic field coils, the dashed line is the vacuum boundary, and the dash- dotted line is the axis of symmetry.

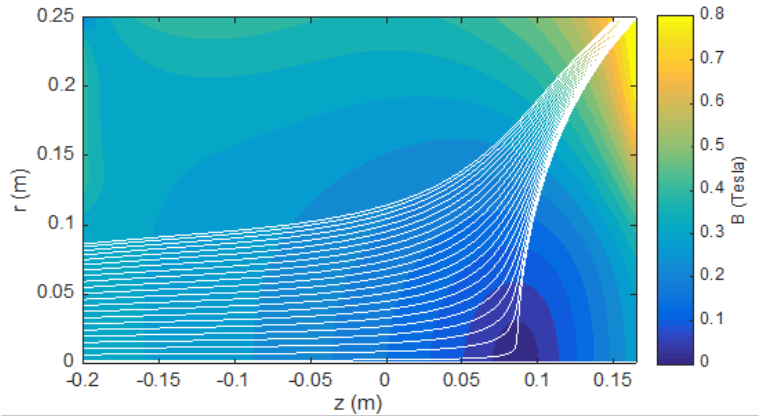
1.3 Progress Steps

After further investigation, plans to build a version of the device were initiated by a team at Princeton University. Before this summer project began, they had already partially constructed the device, having several coils in place already, and some limitations on the possible current values that could be attained. To come up with a configuration for the coils of the device, Matthew Galante wrote a program in python using a Biot-Savart solving technique and streamlines to create visual representations of the magnetic field lines and field strength.



Some preliminary coil placements were determined using this program, as above.

While some coil configurations may initially have looked promising because of how sharply the field lines converged, when a color gradient of field strength is superimposed as below it becomes clear that the mirror ratio is only about 4. and thereby possibly insufficient to trap the particles at high v_i.



To address this potential issue, this project's focus was to write improved code for simulating the magnetic field produced by a set of coils which could eventually be used as a part of an optimization process (gradient descent/ascent) to decide an optimal coil placement for the project.

After producing code which correctly found the magnetic field anywhere in space due to a set of coils (in the form of grouped rings) using the analytic solution of magnetic vector potential for a ring and a curl function, the question quickly became deciding on a cost function to optimize. Simply optimizing the relative ratios and Ian Ochs initially suggested numerically integrating over a 2d Maxwellian to use the ratio of the volumes (which simplifies to the areas) of the loss cones for each side of the device in a separation equation of the form $S = (f_H - f_L)^2$.
However, Professor Fisch expressed his doubts as to how accurate of a predictor the loss cone ratio would be for the escape ratios of a collisional plasma, since particles may jump into or out of the loss cone while on the edge and undergoing collisions, due to the change of direction undergone.

On the topic of escape ratios for a machine with different loss cones on either side, little literature seemed to exist. Specifically in reference to the MCMF, Fetterman, in chapters 7 and 8 of his thesis, addressed this very same problem [2] Unfortunately, since he analytically evaluates the separative power as a function of the loss cone using an idea similar to that proposed by Ochs, Fisch's objection still applies. To address this dirth of information, this project's second main component began. While it began as a search for a metric one could use as a cost function for the coil placement optimization project (while also having some kind of predictor of the separation ratio), we set out to write a program to more generally simulate the motion of particles in a machine with

different loss cones on either end (in this case the result of two different mirror ratios) while the particles undergo collisions. With such a program, we would simply analyze how the escape ratios and loss cone ratios scale to hopefully be able to better estimate expected separation power in the machines it seeks to emulate.

2 Simulation Code

The simulation code was written in Julia, a relatively new programming language with performance approaching that of C. It used the Distributions package during the implementation of collisions. The Gadfly package was used to graph things while debugging and testing the initial particle motion simulator.

Before progressing to the final iteration of the simulator, code was initially written to simulate particle motion in electromagnetic fields before progressing to simulation of a mirror machine, implementing collisions, randomizing starting conditions, and finally adding the necessary loops to do hundreds of trials while varying some parameters of the collision and loss cone.

2.1 Initial Steps towards the Simulation

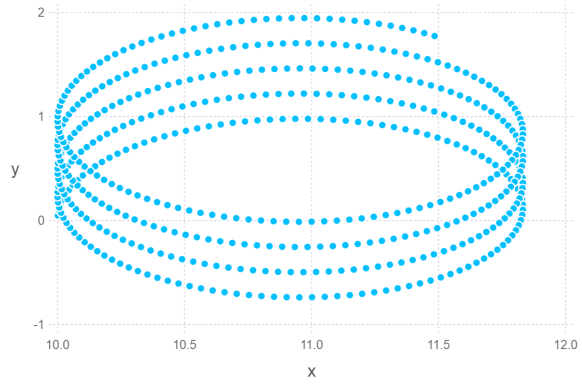
Since the code is a bit complicated and was developed in several steps, it will be easier to first explain the basic particle motion simulator before going on to the rest of the project.

The particle has a 6-dimensional vector describing its position and velocity in three dimensions. It also has an associated mass and charge. The space in which it exists can have a defined electric field and magnetic field function, although in the final simulator, only the magnetic field was relevant. Additionally, there is a function which finds the acceleration of the particle at a given time using the Lorentz force equation:

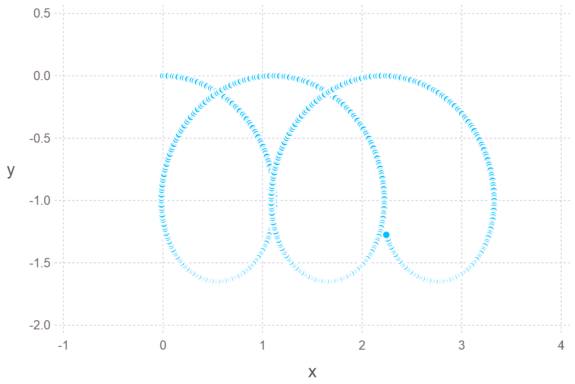
$$F = q(E + v \times B)$$

To make the simulation more accurate, a version of the Runge-Kutta stabilization algorithm for numerically solving differential equations is used, but this will be described more in-depth below. The values of the particle's location can be stored and then plotted. Several examples of it verifying known phenomena are given below.

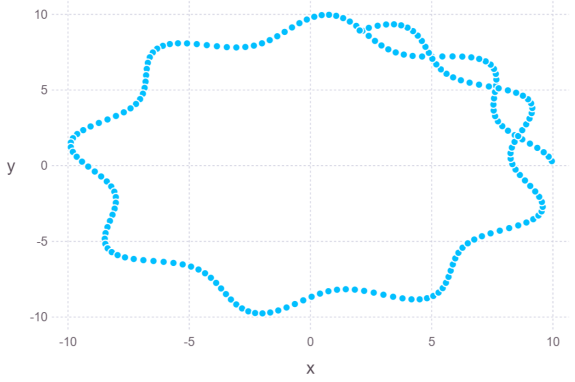
For a positively charged particle with initial x and y velocities in a magnetic field with only a z component which decreases along the x direction, the following drift is observed:



For a similar particle but with a B field in the z direction, and an E field in the y direction, we see the familiar $E \times B$ drift below:



Finally, for a toroidal magnetic field, we see the following particle motion:



With a working particle motion simulator, the rest of the code is a matter of simulat-

ing a mirror-machine-like magnetic field structure, implementing collisions, randomizing starting conditions, and adding loops to vary conditions of the simulation and iterate trials.

2.2 Pitch Angle Scattering Description

For this simulator we used pitch-angle scattering collision. Essentially, at each time step a random angle is drawn from a distribution and the particle's angle is adjusted by this amount. The standard deviation is decided by a value ν or equivalently $\tau = 1/\nu$, the average time it takes a particle to diffuse one radian. Much of the following section was originally drafted by Ian Ochs to help clarify the implementation of these collisions.

2.2.1 Pitch angle

The pitch angle is the relative angle between the magnetic field and the particle velocity. Thus a particle with a pitch angle of 0 travels in a straight line parallel to the magnetic field, while a particle with a pitch angle of $\pi/2$ completes circular orbits with no parallel velocity. In general:

$$\alpha = \arccos\left(\frac{v_{\parallel}}{v}\right) = \arccos\left(\frac{\mathbf{v}\cdot\hat{b}}{v}\right) \tag{2.1}$$

In a pitch angle scattering interaction, the pitch angle changes, while the magnitude of v does not.
Thus, after the collision, $\alpha \to \alpha'$, and we have

$$v'_{\parallel} = v\cos\alpha' \tag{2.2}$$

$$v'_{\perp} = v\sin\alpha'. \tag{2.3}$$

2.2.2 Collision frequency

Because collisions in plasmas tend to take the form of many small-angle scattering events, it can be treated as a random walk.
Thus the collision frequency ν is the inverse of the average time it takes a particle to diffuse 1 radian in pitch angle.
Since in a random walk, the variance scales linearly with the time, the standard deviation scales as $\sqrt{t}$.
After a time $\tau = 1/\nu$, we expect the particle to have diffused 1 radian. Combining these facts, we have:

$$SD(\alpha) = \sqrt{t\nu}. \tag{2.4}$$

Note that we can relate this more conventional collision frequency to the 90° collision frequency by

$$\nu_{90} = \frac{4}{\pi^2}\nu. \tag{2.5}$$

2.3 Implementation

Implementing pitch angle scattering has three mains steps:
First, we must transform to the coordinates in which the pitch angle is defined.
Then we must perform the rotation in these coordinates.
Finally, we transform back.

2.3.1 Local Coordinate System

Our local coordinates are $\hat{b}$ and

$$\hat{v}_\perp = -\frac{(\mathbf{v} \times \hat{b}) \times \hat{b}}{|(\mathbf{v} \times \hat{b}) \times \hat{b}|} \tag{2.6}$$

which points along the component of velocity perpendicular to $\hat{b}$.

We then have

$$v_\parallel = \mathbf{v} \cdot \hat{b} \tag{2.7}$$

$$v_\perp = \mathbf{v} \cdot \hat{v}_\perp \tag{2.8}$$

from which we can calculate α using Eq.(2.1). Note that actually calculating $v_\perp$ is unnecessary.

2.3.2 Scattering rotation

To draw an angle using Julia's distributions package, we use a bounded normal distribution centered around α with standard deviation $\sqrt{dt\nu}$.

From this we will want to draw a random variable α'

Note that we want our timestep to be at most some fraction of $1/\nu$. Thus overall we have

$$dt = \epsilon \min\left(\Omega^{-1}, \nu^{-1}\right) \tag{2.9}$$

where epsilon is small compared to 1.

2.3.3 Reverse transformation

Finally, we must convert back to our original coordinates. Since we already calculated the relevant basis vectors, we simply have:

$$\mathbf{v} = v_{\parallel}\hat{b} + v_{\perp}\hat{v}_{\perp} \tag{2.10}$$

2.4 Code Explanation and Outline

2.4.1 Initial Conditions

Initial conditions that can be varied by the user include the initial magnitude of the velocity vector, the mass of the particle, the time step of the simulation, the charge of the particle, an electric field, and the initial value of the magnetic field.
Additionally, the simulation loops through a set of τ values which vary the frequency of collisions, these can be adjusted manually, and as many τ values as desired can be entered, as they comprise the outer loop of the simulation. Immediately within this loop is another loop which varies the outer strength of the magnetic field. This is also easily manually adjustable. Finally, there is a loop which runs through a number of trials which can be set by the user.

2.4.2 Magnetic Field Function

Within the loop varying the magnetic field strength on one side of the machine, we first define our magnetic field function. Ian Ochs recommended using the function $B_z = B_0(1 + z^2)$ so that the z component gets stronger along the z axis in the way a mirror machine requires.
To find x and y components which correspond to a realistic mirror machine, we use Maxwell's equation based on the divergence of a magnetic field.

$$\nabla \cdot B = 0$$

$$\frac{\partial B_x}{\partial x} + \frac{\partial B_y}{\partial y} + \frac{\partial B_z}{\partial z} = 0$$

We will also make $B_x = B_y$ so that it is symmetrical about the z axis.

$$2\frac{\partial B_x}{\partial x} + \frac{\partial B_z}{\partial z} = 0$$

$$2\frac{\partial B_x}{\partial x} + 2B_0 z = 0$$

$$\int \frac{\partial B_x}{\partial x} dx = -\int B_0 z dx$$

$$B_x = -B_0xz + C(y,z)$$

We want the simplest solution, so we choose $C(y,z) = 0$ (and for B_y, we equivalently choose $C(x,z) = 0$) giving

$$B_x = -B_0xz$$

and

$$B_y = -B_0yz$$

Unfortunately, this only gives us a function which scales up the magnetic field as we move further along the z axis. To scale the z axis down such that the magnetic field value which would be at "c" instead occurs at $x = 1$ we change our final expression for B to the following:

$$B = [B_x, B_y, B_z] = [-B_0xzc^2, -B_0yzc^2, B_0(1 + (zc)^2)]$$

We can then use this magnetic field function with the earlier described Lorentz force equation to get acceleration values $a = \frac{q}{m}(E + v \times B)$

2.4.3 RK46-L Stabilization Algorithm

When numerically solving differential equations the errors can compound and become very large over time. The fourth-order six-stage Runge-Kutta algorithm as proposed by Julien Berland et al. in their paper "Low-dissipation and low-dispersion fourth-order Runge-Kutta algorithm" was used to reduce this numerical error in our particle simulation [3]. In their paper they provide the coefficients and constants used in implementation.

We stored two sets of values labeled α and β, as in the paper, and then in the time step loop we went through the following process:
Notationally, let $f[n:m]$ refer to the values n to m of an array. We looped through each set of α and β values (there are six, one set for each stage) then, given our six-vector $p[1:6] = [x, y, z, v_x, v_y, v_z]$, our time step dt a new value $\omega = 0$, and our acceleration function $accel()$, we set a new set of values

$$F[1:3] = p[4:6]$$

$$F[4:6] = accel(p)$$

Then we set

$$\omega' = \alpha_i\omega + F[1:6]dt$$

and finally

$$p' = p + \beta_i\omega$$

completing the loop. After all 6 sets of α and β values are looped through, the time step has been incremented. For more detail on this process, see the original paper.

2.4.4 Simulation Loop Starting Conditions

As we would not want our data biased by the direction in which the particle's velocity begins, immediately within the loop that loops through the simulation for each trial, we randomize the starting velocity as follows.
We begin with the magnitude of the velocity V as chosen at the beginning of the code. We then choose a random angle θ from a random uniform distribution between 0 and $\pi/2$.
We then choose a random bit from Julia's "bitrand" function that will be used to decide the sign of the direction of the z component. Note that we ignore the y components and the sign of the x component due to the azimuthal symmetry.
We set

$$p = [0, 0, 0, V\cos\theta, 0, V\sin\theta]$$

and then finally we have an "if" statement such that if "bitrand" gave a 1 we multiply $p[6]$ by -1.

2.4.5 Simulation Loop

In the loop which simulates the motion of the particle, we begin each time step by incrementing based on the RK scheme, then adjust it based on the pitch-angle scattering collision, and then check to see if we have crossed the boundaries 1 and -1, if so, we adjust some counters that have been put in the simulator to keep track of how many particles leave on either side. Eventually these counters are divided by the trial count to get the ratio of particles that leaves on either side. (At this point in the code, there are a few statements commented out that were used for debugging purposes. They were used to graph the particle's z coordinate over time. They will be left there in case they are useful for anyone who wishes to edit or re-purpose the code.)

At the end of the code, we print the ratio of the loss cones (which we calculate briefly near the end of the code), the escape ratios, and some values we have included to try to characterize each simulation trial in terms of the particle's associated values, the collision frequency, the size of the "mirror machine," et cetera.

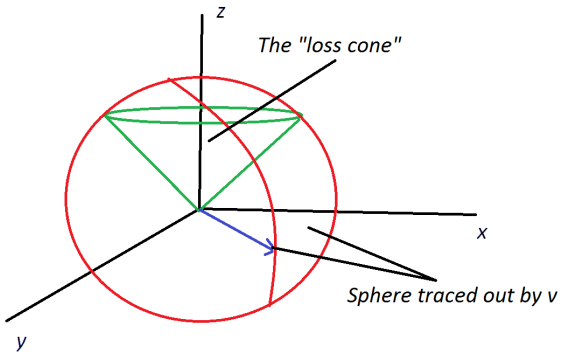
The code can be found in its entirety in Appendix I near the end of the paper.

3 Loss Cone Theory Predictions

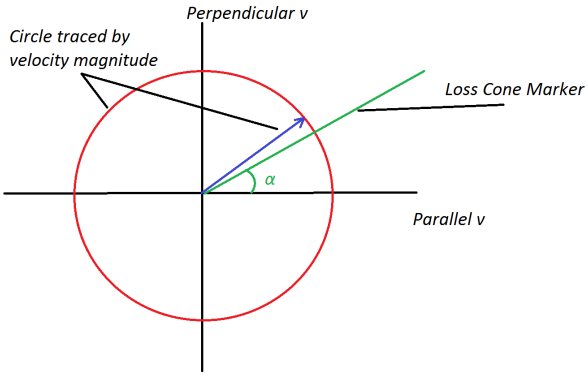
We originally supposed that the escape ratios would scale with the loss cone, but eventually reevaluated and realized that they may scale with the log of it, or the square root of it, or some other variation. It also may vary depending on the collision frequency of

the particles in the plasma. Because of this, we first briefly show how we will address the loss cone areas.

Because our scenario is azimuthally symmetric, the 3d loss cone



can be reduced to a 2-dimensional representation, seen below.



Note that α is sometimes instead measured from the perpendicular axis.

In our case, we let $\alpha = \arctan\left(\frac{1}{\sqrt{\frac{B_{max}}{B_{min}} - 1}}\right)$

We use this to find the loss cone area on either side of our simulation (since each should be measured from the center, where B_{min} exists).

We can simplify the ratio of the areas to the ratio of the areas in one circle quadrant, since it scales to the whole circle. Each circle sector can be written as $A = r\theta$ but since the radius for each sector is the same, they cancel out when written as a ratio. Thus we let the ratio of the loss cone areas be

$$LR = \frac{\alpha_2}{\alpha_1 + \alpha_2} \tag{3.1}$$

where α_1 is the side with a weaker magnetic field, the one kept constant.

4 Simulation Results

5 Field Simulation and Optimization for the MCMF

5.1 Code Explanation and Outline

The code for this program was written in Matlab. It was written to improve on the Python code written by Matthew Galante by using the analytic expression for the curl of a magnetic field at a point due to a ring instead of using the Biot-Savart law, and is intended to be used in a loop to optimize the placement of the coils in the MCMF.

The program has three main parts:
A function which expresses the magnetic vector potential at a point.
A function which numerically takes the curl of a function in cylindrical coordinates.
And the main part, which keeps track of the values of B at every point, then uses Matlab's "quiver" function to plot the magnetic field lines, and the "contour" function to plot the field density throughout space.

5.1.1 Magnetic Field Simulation

The first function that we used was one to find the magnetic vector potential at a point due to a circular current loop analytically using an expression given by James Simpson et al. in their paper "Simple Analytic Expressions for the Magnetic Field of a Circular Current Loop." [4]
For a current loop with radius a and current I lying on the $x-y$ plane, they give the expression with at a point (r, θ, ϕ) as follows

$$A_\phi(r,\theta) = \frac{\mu_0 I a}{\pi} \frac{1}{\sqrt{r^2+a^2+2ar\sin\theta}} \frac{1}{k^2} \left[(2-k^2)K(k^2) - 2E(k^2)\right],$$

where

$$k^2 = \frac{4ar\sin\theta}{r^2+a^2+2ar\sin\theta},$$

and $K(k^2)$ and $E(k^2)$ are the complete elliptic integrals of the first and second kind, respectively, defined by

$$K(k^2) = \int_0^{\pi/2} \frac{1}{\sqrt{1-k^2\sin^2\alpha}} d\alpha, \quad E(k^2) = \int_0^{\pi/2} \sqrt{1-k^2\sin^2\alpha}\, d\alpha.$$

The cylindrical curl function allows input of the size of the differentials dr and dz. We note that the magnetic vector potential of a ring has only a $\hat{\theta}$ component, and thus we can find the curl as

$$\nabla \times A = -\frac{\partial A}{\partial z}\hat{r} + \frac{1}{r}\frac{\partial (rA)}{\partial r}\hat{z}$$

We therefore take the partial derivatives of A with respect to z and r as

$$\frac{\partial A}{\partial z} = \frac{A(z+dz) - A(z)}{dz}$$

and

$$\frac{\partial (rA)}{\partial r} = \frac{(r+dr)A(r+dr) - rA(r)}{dr}$$

Making our magnetic field as a function of the magnetic vector potential

$$B = [B_r, B_z] = [-\frac{\partial A}{\partial z}, \frac{1}{r}\frac{\partial (rA)}{\partial r}] \tag{5.1}$$

Finally, the main piece of the code, labeled "FieldPlot" in Appendix II, begins by allowing the user to input a group of coils as a 3xN array with coil entries as (I, a, z). While the functions above were in spherical coordinates, we use Cartesian coordinates here to reconcile having many coils along the z axis and to allow us to use the streamline function, since trying to do this in cylindrical would not let us store the values at evenly spaced (x, y, z) coordinates. The user can also change the increment by which the coordinates should be scaled, and the amount of space they wish to be stored.

The code continues by creating a matrix to store the B_r and B_z values, then looping through each ring and adding to the initialized matrix. For each ring, it takes the curl of A at the each point in space then adds to the existing B_r and B_z values at the point. Then the program does some scaling before making the contour and quiver plots of the data.

Some plots generated by this code can be found in the section Results, and the entire code can be found in Appendix II.

5.1.2 Optimization Process

Using a cost function based on factors including the loss cone ratios due to the centrifugal containment condition and the mirror ratio containment condition for the MCMF, we aim to implement a simple gradient ascent algorithm while varying the coil placement and potentially the currents running through the coils to optimize the cost function. We will numerically calculate the gradient induced by a slight variation of these variables and repeatedly vary them until reaching a local maximum of the cost function, and also repeat this with several sets of starting functions, since there may be many relative maxima

but we want to find the absolute maximum. This should be relatively straightforward once the cost function is finalized, and even without a reliable predictor of the actual separative power of the MCMF, we can still attempt to optimize the ratios based on the loss cones, since this should at the very least be highly correlated with the loss ratios as evidenced by the particle escape/collision simulations described in the first section of the paper.

5.2 Results

The following are the set of things generated by the programs created, with a brief explanation of the results accompanying them.

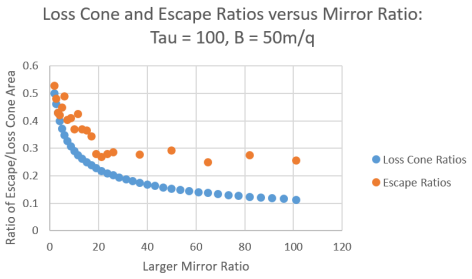
The following graphs were generated by the escape ratio simulator. The first three left graphs have both the ratio of the loss cone areas and the escape ratios graphed, but since this doesn't give much insight into the relation between the loss cone and escape ratios, similar graphs are left off for the other τ values. The simulations here had 500 trials each with .001 second time steps. Simulations were done in sets based on the magnetic field and velocity values. For the first 13 graphs the velocity was set to 1500m/s, for the penultimate set of graphs it was $100m/s$, and last set of graphs it was set to only $2m/s$. We scaled down the velocity so much since the boundaries used were 1 and -1, and since $vdt > 1$, particles that got a velocity with a large parallel velocity in either direction need only one time step to escape. While this doesn't completely invalidate the data, it makes it much more random and difficult to evaluate. For this reason the final set of trials is in a regime such that the underlying effects are more apparent.
Each set of trials is accompanied by several values - such as the gyrofrequency and bounce time - given to help characterize the mirror machine, velocity, and magnetic field.
The gyrofrequency is $\omega_g = \frac{|q|B}{m}$
The bounce time is given as twice the length of the machine divided by the velocity of the particle.
Note that in choosing these values in the simulation, we wanted to choose values that make sense in the real world. This is why the value $v = 1500m/s$ was used intitially, since $v = 1500m/s$ corresponds roughly to a helium atom at .1 eV, since the thermal velocity

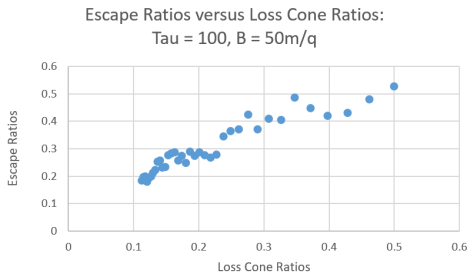
$$v_t = 10^4 \sqrt{\frac{E_{eV}}{M_{amu}}}$$

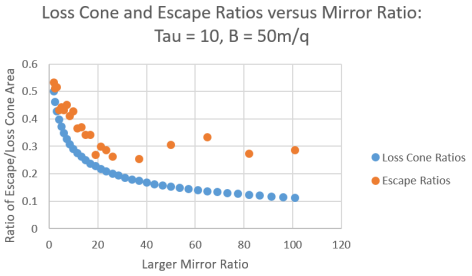
We also should have made sure to keep $\omega_g >> 1/$bounce time, and the gyroradius small compared to the device volume. To balance these values, we also needed to keep the magnetic field strong enough to be consistent and keep our time step size small relative to $1/\omega_g$.

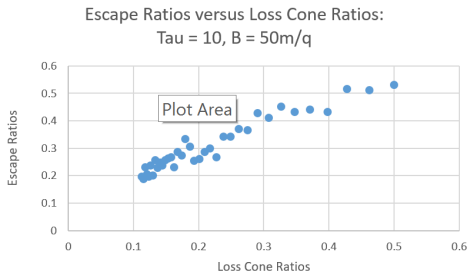
The data obtained is as follows:

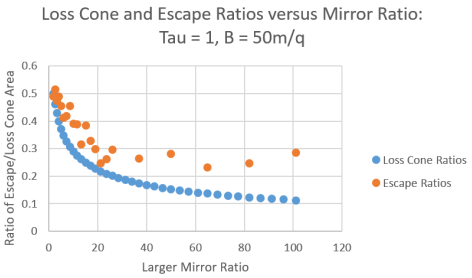
$1/\omega_g = 0.02$
bounce time $= .002667$
time factor $= (1/\omega_g)(\text{bounce time}) = 5.3e-5$

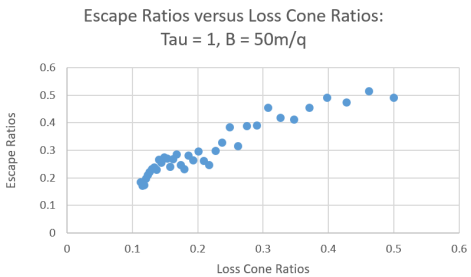


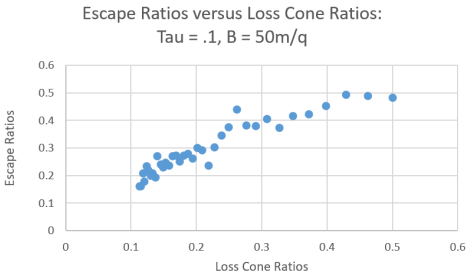


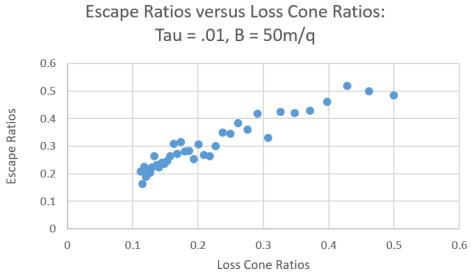




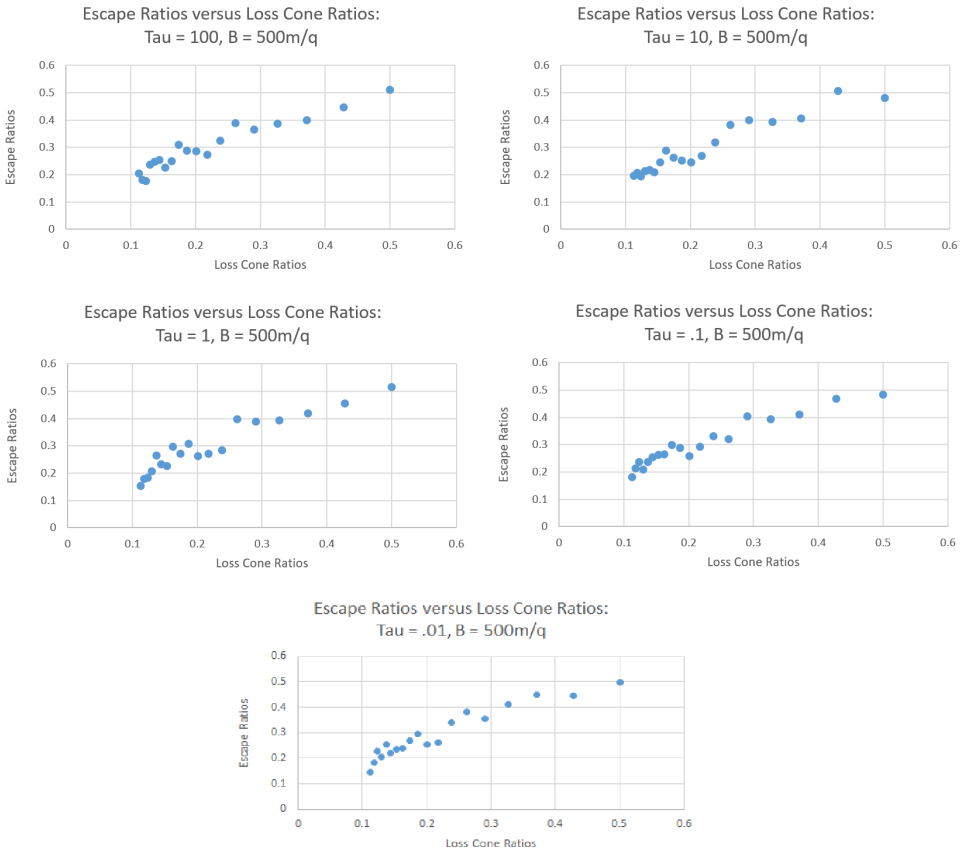




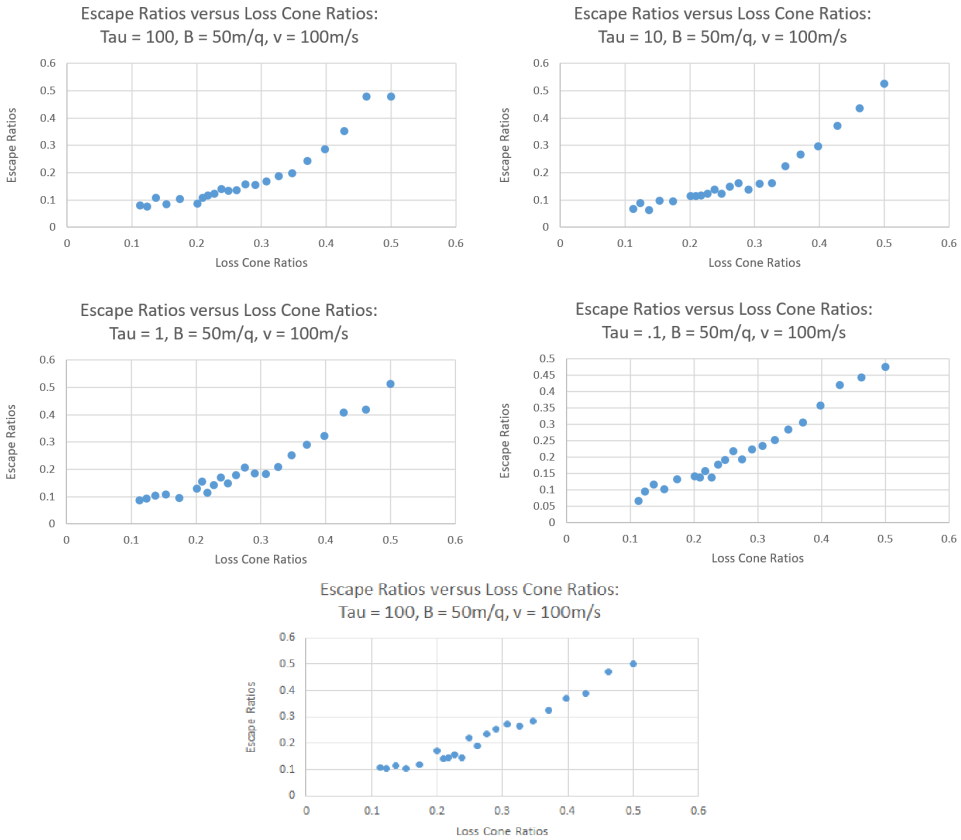




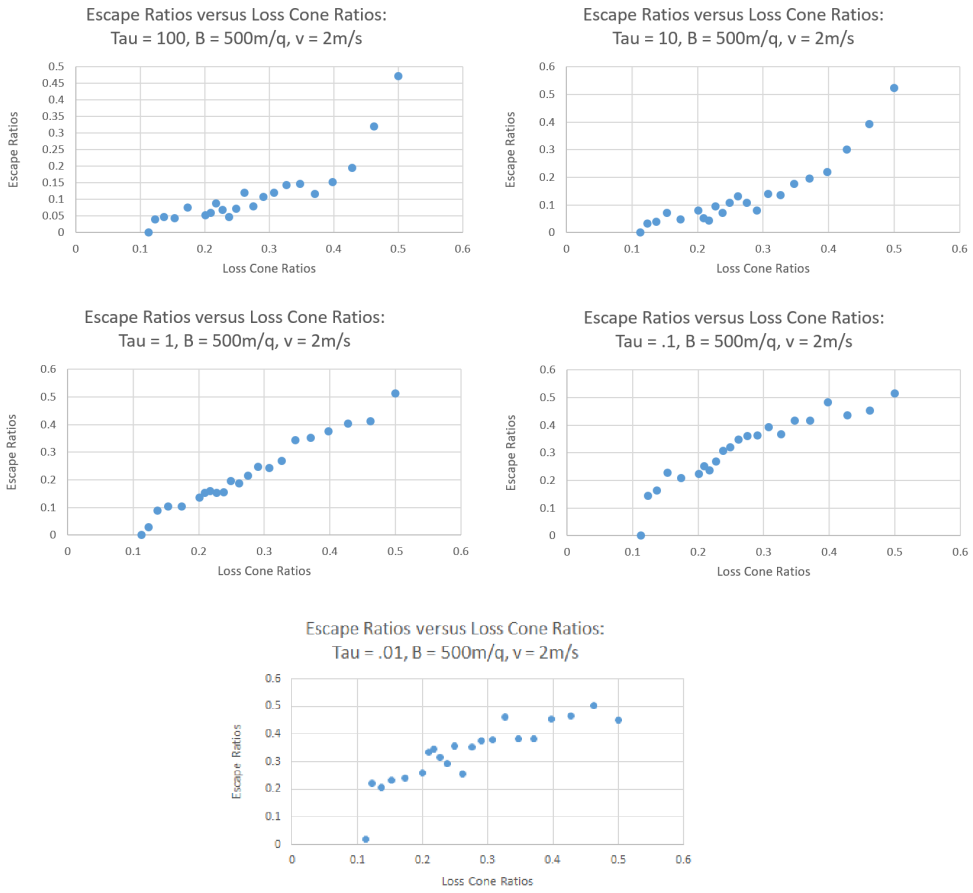
$1/\omega_g = 0.002$
bounce time $= .002667$
time factor $= (1/\omega_g)(\text{bounce time}) = 1.3e-6$



$1/\omega_g = 0.02$
bounce time $= 0.04$
time factor $= (1/\omega_g)$(bounce time) $= 0.0008$

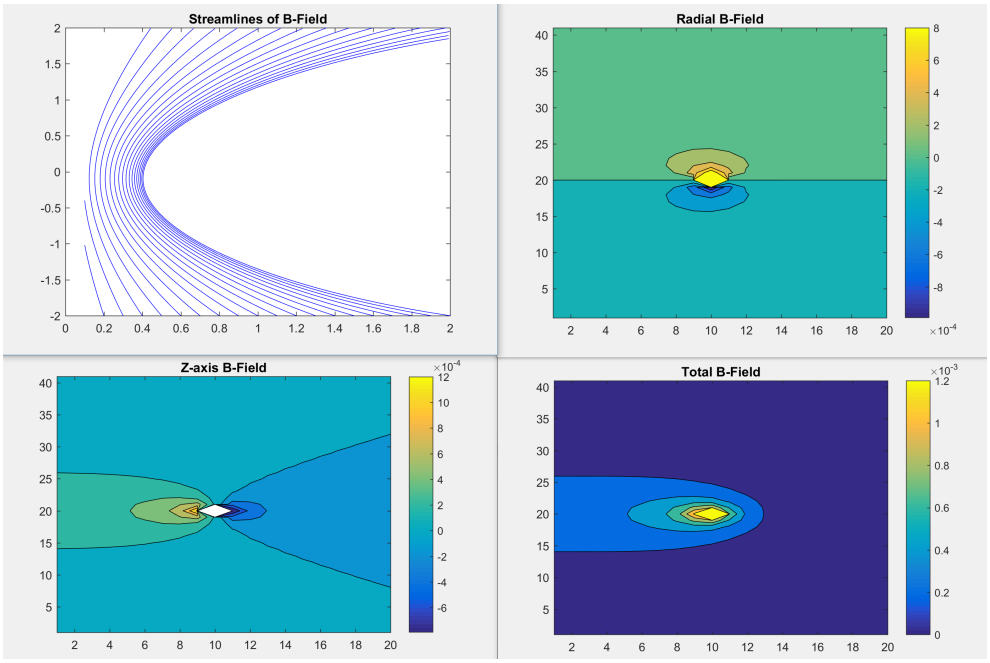


$1/\omega_g = 0.02$
bounce time $= 2.0$
time factor $= (1/\omega_g)$(bounce time) $= .04$

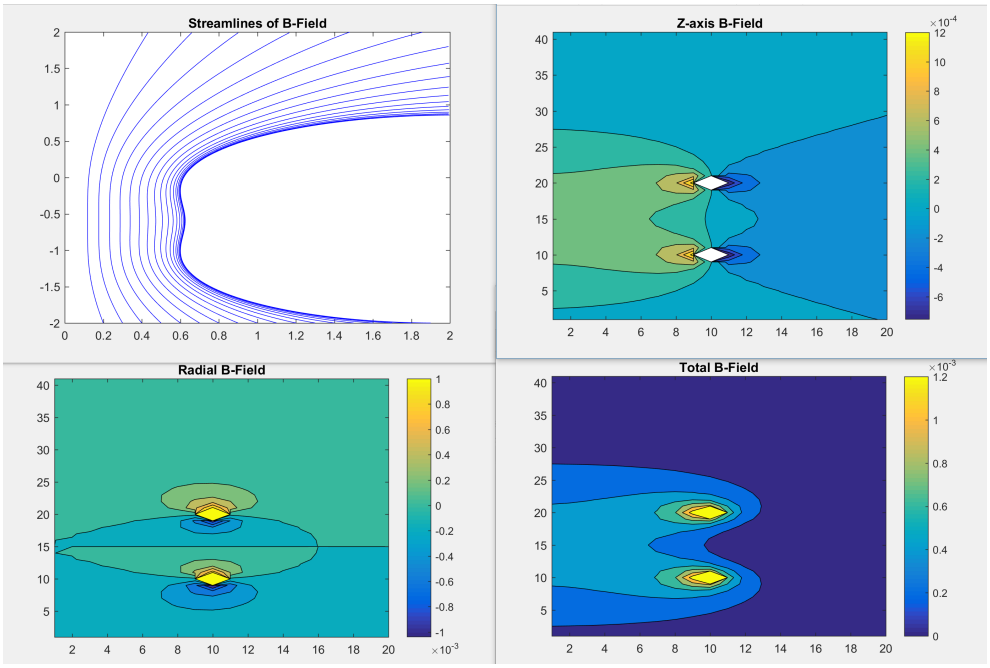


From these plots, it appears that with high collision rates, as with higher velocity values, the escape ratio and loss cone areas appear to scale in a more linear fashion, but when the velocity and collision rates are lowered, the particles very quickly shift so that the ratios do not scale linearly. Instead, the escape ratio changes much more quickly, and the particles leave much more heavily through the lower-mirror-ratio exit than before. From this, we see that given certain conditions, simply using the loss cone as a metric to determine separation of the particles may be reasonable, but it will likely not scale linearly with the loss cone area. Also, it seems like the realm of particle speeds, magnetic field, and collision frequency all can potentially have significant effects on the particles' escape characteristics. Further simulations should be run with different time steps and velocity values to see how these values may be related to the escape of the particles.

Some preliminary plots and data generated by the magnetic field generating program are shown below, including coil arrangements similar to those which should be used in the MCMF. The first image is simply a plot based on one coil.

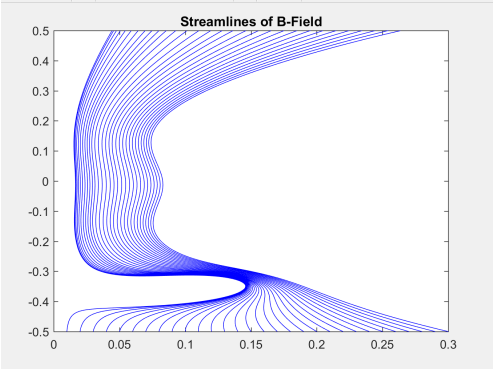


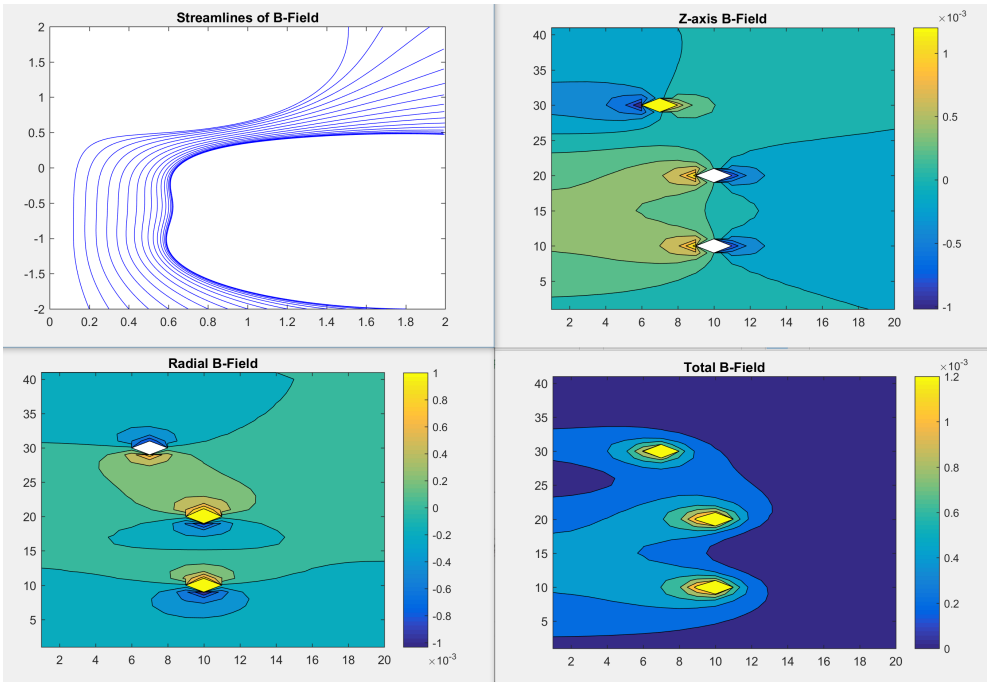
Here is a plot with two identical coils placed next to one another.



The following two plots have three coils, but one of them has the current running in the opposite direction of the other two. These configurations generate fields similar to that

of Galante's Python code as shown in the introduction, and are along the same lines as the idea on which the MCMF's coil placement is based.





6 Appendix I - Particle Simulation Code

```
# Repeatedly simulates motion of an ion in a mirror machine.
# Has adjustable z-bounds to adjust mirror ratio.
# Iterates the process to determine loss ratio.

using Distributions
using Gadfly # Only for the plotting/debugging


# Keeping track of ratios
LC = [] # Loss cone areas
ER = [] # Escape ratio


# Boundary conditions. Don't adjust these.
zBound1 = 1
zBound2 = -1

# Initial conditions.
V = 1500                    # magnitude of velocity vector
m = 1.67e-27                # mass of a proton in kg
t = 0.0                     # current time, starts at 0
dt = .001                   # change in time in seconds
q = 1.60e-19                # charge of a proton in coulombs
E = [0, 0, 0]               # electric field in V/m
B_0 = 50*m/q
trialcount = 75

# Time normalization
bounceFreq = (zBound1 - zBound2)/(.5*V)
inverseGyroFreq = m/(q*B_0)
timeFactor = bounceFreq*inverseGyroFreq

# Loop through tau values
for i = [100, 10, .1] # [100, 10, 1, .1, .01]
    tauValue = string("tau = ", i) # (i/timeFactor), "*(time scale factor)")
    push!(ER, tauValue)
# Loop through boundary conditions
    for j = [1:1:10;] # [1:.25:5, 6:10;]

        Bound1Counter = 0
        Bound2Counter = 0

# Collision conditions/distribution.
tau = i     # average time it takes a particle to diffuse 1 radian in pitch angle
            # should be greater than m/q*B (which is 1 over the coefficient of B_0 below)
            # how much of a difference should I have?
```

```
nu = 1/tau  # inverse of tau - collision frequency
angles = Normal(0, sqrt(dt*nu))


# Function that simulates B-Field
function B(x)
    B_0 = 50*m/q
            if (x[3] < 0)
    B_x = -B_0*x[3]*x[1]
    B_y = -B_0*x[3]*x[2]
    B_z = B_0*(1+x[3]^2)
            end
            if (x[3] >= 0)
    B_x = -B_0*(x[3]*j*j)*x[1]
    B_y = -B_0*(x[3]*j*j)*x[2]
    B_z = B_0*(1+(x[3]*j)^2)
            end
    [B_x, B_y, B_z]
end


# f = [] # x-axis if I'm graphing.
# g = [] # y-axis if I'm graphing.


# Function that simulates Lorentz-force-based acceleration.
function accel(x)
            (q/m)*(E + cross(x[4:6], B(x)))
end


# Constants/values for iterations of the RK46-NL scheme
alpha = [0.0, -.737101392796, -1.634740794341, -.744739003780, -1.469897351522, -2.813971388035]
beta = [.032918605146, .823256998200, .381530948900, .200092213184, 1.718581042715, .27]
omega = 0
counter = 1
F = [3.14, 1.59, 2.65, 3.58, 9.79, 3.23]     # These values don't matter.


# Main Loops:
trialCounter = 0

# Iterating the particle motion.
        while (trialCounter < trialcount)

    # To keep track of when to break the loop.
    CurrentBoundOne = Bound1Counter
    CurrentBoundTwo = Bound2Counter
```

```
# Randomizing initial velocity conditions.
initialAngle = rand(Uniform(0, pi/2))
signs = bitrand(2)
p = [0, 0, 0, V*cos(initialAngle), 0, V*sin(initialAngle)]
    if (signs[1])
        p[4] = -1*p[4]
    end
if (signs[2])
        p[6] = -1*p[6]
    end

# Loop while within the bounds.
temp =
while (CurrentBoundOne == Bound1Counter) & (CurrentBoundTwo == Bound2Counter)

    # RK adjustments.
    while counter <= 6
    F[1:3] = p[4:6]
    F[4:6] = accel(p)
    omega = alpha[counter]*omega + dt*F
    p += beta[counter]*omega
    counter += 1
    end


# Collision occurs
Bf = B(p)
totalB = norm(Bf)
bUnit = Bf/totalB                                   # Unit vector in b direction
crossP = cross(p[4:6], bUnit)
totalProduct = norm(cross(crossP, bUnit))
vPerpUnit = -(cross(crossP, bUnit))/totalProduct  # Unit vector in v_perp direction.
    if ((dot(p[4:6], bUnit) > norm(p[4:6]) ) | (dot(p[4:6], bUnit) < -norm(p[4:6]) ))
                println("error")
            end
                v_par = dot(p[4:6], bUnit)
alpha_i = acos(v_par/norm(p[4:6]))
v_perp = dot(p[4:6], vPerpUnit)
random = rand(angles)
alpha_f = alpha_i + random
v_par = norm(p[4:6])*cos(alpha_f)
v_perp = norm(p[4:6])*sin(alpha_f)
p[4:6] = v_par*bUnit + v_perp*vPerpUnit
p[4:6] = V*p[4:6]/norm(p[4:6])


    # Checking if we are outside boundaries and adjusting counters accordingly
        if (p[3] >= zBound1)
        Bound1Counter = Bound1Counter + 1
```

```
            end

            if (p[3] <= zBound2)
            Bound2Counter = Bound2Counter + 1
            end
        # Incrementing time and resetting RK adjustment counter.
    t = t + dt
    counter = 1
       # push!(f, t)    # graphing the time axis. For debugging mainly.
       # push!(g, p[3]) # graphing the z-axis.
    end

    trialCounter += 1
end

        tempLC = atan(1/sqrt((B([0, 0, 1])[3]/B([0, 0, 0])[3]) - 1))
        if (tau == 10)
        push!(LC, (tempLC/(tempLC + .7853982)))
        end
        push!(ER, Bound1Counter/(Bound1Counter + Bound2Counter))
    end
end

println("Loss Cone Ratios:")
for i in LC
    println(i)
end
println()
println("Escape Ratios:")
for i in ER
    println(i)
end
println("")
println("1/gyrofrequency: ", inverseGyroFreq)
println("bounce time: ", bounceFreq)
println("time factor = (1/gyrofrequency)*(bounce time) = ", timeFactor)
# plot(x=f, y=g, Geom.line) # This prints a graph of the positions. Used mainly for debugging.
# Uncomment lines 39, 40, 121, 122, 138 to graph z versus t.
```

7 Appendix II - Field Simulation Code

mVectPot.m

```
function A = mVectPot(I,a,r,z)
%mVectPot
%   Takes the current, radius of the ring, distance r away, and altitude
%   angle elevation and outputs the magnetic vector potential at the point.

% Converting coordinates cylindrical to spherical
rho = sqrt(r*r+z*z);
elevation = atan(r/z);

if elevation < 0
    elevation = elevation + pi;
end

mu_0 = 4*pi*(10^-7);
denom = rho^2 + a^2 + 2*a*rho*sin(elevation);
kSquared = 4*a*rho*sin(elevation)/denom;

[K,E] = ellipke(kSquared);

A = (mu_0*I/pi)*(1/sqrt(denom))*(1/kSquared)*((2 - kSquared)*K - 2*E);
end
```

cylindricalCurlPart.m

```
function B = cylindricalCurlPart(fname,I,a,r,z)
% This is a diminished version of a curl operator that only works for
% azimuthally symmetric functions

% Enter size of differentials
dz = .0005;
dr = .0005;

partialAz = (fname(I, a, r, (z + dz)) - fname(I,a,r,z))/dz;
partialrAr = ((r + dr)*fname(I, a, (r + dr), z) - r*fname(I, a, r, z))/dr;
Br = -partialAz;
Bz = (1/r)*partialrAr;
B = [Br, Bz];
end
```

FieldPlot.m

```
% A program which plots the magnetic field lines of a group of current
% carrying rings which are specified. Also creates contour maps of the
% total field strength, and of each component.

% Variables to Provide:
```

```
% Store coils as rings in a 3xN array (I, a, z)
% Note: put backwards current below other coils to start streamlines below
Coils = [300, .2, -.15; 300, .2, .15; -20, .02, -.25; -50, .12, -.33];

N = size(Coils,1); % number of rings - temporarily manual entry
zMetricSize = 1;   % How large the z-axis should be in meters
rMetricSize = .3;   % How large the r-axis should be in meters
zScale = .01;       % The increment that z samples will be taken
rScale = .01;       % The increment by which r-samples will be taken
startZshift = -.5;  % At which z-coordinate the streamlines begin



% Scaling everything for array use
zSize = zMetricSize/zScale;    % number of array elements needed to store z
rSize = rMetricSize/rScale;    % number of array elements needed to store r
zShift = -(zSize/2);           % Since I'm centered at z = 0,
                               % I want half the points below

% Making an array to store Br and Bz
B = zeros(zSize + 1, rSize, 2);

% Loop through r and z to fill arrays with B values
for i = 1:zSize
    z = (i + zShift) * zScale;

    for j = 1:rSize
        r = j * rScale;
% Loop through rings

        for k = 1:N
% Take the curl of A at the point, add components to the B array

            if r == 0             % Taking care of a corner case when r = 0
                                  % (This actually isn't relevant for now
                                  % since I don't start at 0. But it
                                  % doesn't hurt to leave it in case.)
temp = 4*pi*(10^-7)*Coils(k, 1)*(Coils(k, 2)^2)/(2*((z - Coils(k, 3)^2 + Coils(k, 2)^2)^1.5))
Btemp = [0, temp];
            else
Btemp = cylindricalCurlPart(@mVectPot, Coils(k, 1), Coils(k, 2), r, (z - Coils(k, 3)));
            end

            B(i, j, 1) = B(i, j, 1) + Btemp(1);   % r-coordinate of B
            B(i, j, 2) = B(i, j, 2) + Btemp(2);   % z-coordinate of B
        end
    end
end
```

```
% Making r and z coordinates for plotting, scale them.
r = [1*rScale:rScale:(rSize*rScale)];
z = [zShift*zScale:zScale:(zSize + zShift)*zScale];
[R,Z] = meshgrid(r, z);

% Contour plots of B (total, radial, z)
totalB = sqrt(B(:,:,1).^2 + B(:,:,2).^2);
figure
contourf(totalB);
title('Total B-Field');
colorbar;

figure
contourf(B(:,:,1));
title('Radial B-Field');
colorbar;

figure
contourf(B(:,:,2));
title('Z-axis B-Field');
colorbar;

% Plotting streamlines
figure
quiver(R,Z,B(:,:,1),B(:,:,2))
startz = startZshift*(ones(size(r)));
streamline(R, Z, B(:,:,1), B(:,:,2), r, startz);
title('Streamlines of B-Field');
```

References

[1] Abraham J. Fetterman and Nathaniel J Fisch *The magnetic centrifugal mass filter* 2011.

[2] Abraham J. Fetterman *Wave-driven rotation and mass separation in rotating magnetic mirrors* 2012.

[3] Julien Berland et al. *Low-dissipation and low-dispersion fourth-order Runge-Kutta algorithm* 2005.

[4] James Simpson et al. *Simple Analytic Expressions for the Magnetic Field of a Circular Current Loop* 2001.